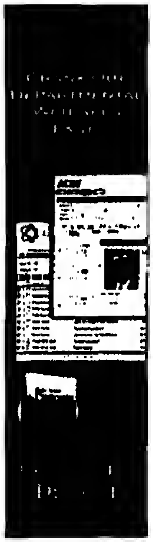
Attachment A

User-Level Spin Locks

By Gert Boddaert.

An introduction to using spin locks for synchronization.

C++ (VC6)
Windows (Win95, NT4, Win2K, WinXP)
Win32, VS
Dev
Posted : 19 Oct 2000
Updated : 24 Oct 2000
Views : 87,785



> "The concept of mutual exclusion is a crucial one in operating systems development. It refers to the guarantee that one, and only one, thread can access a particular resource at a time. Mutual exclusion is necessary when a resource doesn't lend itself to shared access or when sharing would result in an unpredictable outcome... If one thread reads a memory location while another writes to it, the first thread will receive unpredictable data... The same error could occur on a single processor system if the operating system were to perform a context switch while the two threads are accessing this non-shareable resource or critical section... The biggest area of concern is interrupts. The kernel (or a thread) might be updating a global data structure when an interrupt occurs whose interrupt-handling routine also modifies the structure."

Excerpt from the book *Inside Microsoft Windows 2000, Third Edition*, by David Solomon and Mark Russinovich...

1. What is a Spin Lock?

A Spin Lock is an inter-process or task synchronization mechanism like a Mutex, Semaphore or a (Win32) Critical Section. Spin Locks are often used in Symmetric Multiprocessor Operating System Kernels. Because sometimes it is more efficient to constantly poll for the availability of a lock rather then...

1. Pre-empting the thread (doing the context switch), schedule another thread
2. Maintaining and altering the extra housekeeping structures ("who" owns the synchronization object or lock, how long, has the timeout interval elapsed,...)
3. Doing the extra work (of C function calls or even of Assembler routines) for a fair first-in-first-out mechanism.

Compared with the Win32 `CRITICAL_SECTION` structure and its functions, and certainly with any other synchronization mechanism like advanced mutexes and semaphores, a Spin Lock is certainly less fair (there is no protection - in this version anyway - against the theoretical "starvation" of a thread that is forever-waiting for the resource to unlock).

Also, it is more "unsafe" for your programs: if a Thread holding the lock unexpectedly dies (without freeing the lock), the presented Spin Lock has no way of knowing this, hence your threads waiting on the resource will also not be able to recover and thus "spin forever", burning CPU cycles.

But if used very carefully, a Spin Lock is in certain situations a lot faster than any other synchronization mechanism your operating system has to offer (unless it also provides user accessible Spin Locks, of course). These classes (a not-re-entrant and a re-entrant version of a USER-LEVEL SPIN LOCK) were constructed because they are a "semi-portable" synchronization mechanism for threads or tasks within the same process or address space. You can use this lock mechanism in software running on top of an OS, and also on a system without an OS where you have to protect queues, buffers, lists, structures, etc... against interrupt routines, which can leave your shared data in an inconsistent state. This situation has a lot of similarities to a thread synchronization problem (race condition) for a critical section of a standard Win32 multi-threaded program.

All you have to do is to port one (1) function to your specific CPU platform. (The I386 implementation is only 4 instructions long).

When first investigating these synchronization issues for no-OS-Embedded systems, I came across software locking mechanisms such as Peterson's Algorithm (for protecting a critical section against data corruption by 2 competing processes or tasks) and Lamport's Bakery Algorithm (for protecting a critical section against data corruption by N competing processes or tasks). Both solutions (and their derivatives) make the assumption that, reads and writes are "atomic". This just means that, reads and writes happen in a not-interruptible sequence of (CPU) instructions (or preferably only one CPU instruction), which is not the case for most hardware platforms - just take a look a the number of (I386) processor instructions required to store or write a value into a variable.

This explanation brings us to the standard way of avoiding data corruption in the no-OS-world when memory is shared between interrupt routines and a 'normal' - not-interrupt-driven - part of the program (what is mostly referred to as the "main loop"):

=> Temporarily disabling the interrupts from within the main loop when retrieving data from the shared buffer.

Pretty 'lame' - as they say - because this solution can (in extreme cases) lead to the "loss of events", if the 'retrieve' function in the main loop takes too long to complete.

The solution here is to find a processor specific instruction that is SMP safe and executes in one not-interruptible sequence. As an example, the `SWAP` instruction is mostly used. This simple but ingenious idea is, in literature, also referred to as the "Test And Set" method.

2. The Algorithm

The Hardware algorithm for a binary semaphore using the Test and Set method:

```
wait(s)
{
   [illegible]
   {
      set s = 0;
      if (s == 0 && previous value of s == 1)
      {
         break; // we obtained the lock
      }
   }
```

Attachment A

```
    while (true);
}

signal(s)
{
    set s = 1; // release the lock
}
```

Just to set the dot on the 'i':

This is nothing new; you can find this in any good OS internals-explaining textbook.

3. The Spin Lock implementation:

3.1 Non-reentrant spin lock

Collapse

```
    // THIS LOCK IS NOT REENTRANT !!
class CPP_SpinLock
{
public:    // inlined constructor
    // inlined NON virtual destructor
    inline CPP_SpinLock() : m_s(1) {}
    inline ~CPP_SpinLock() {}

        // enter the lock, spinlocks (with/without Sleep)
    // when mutex is already locked
    inline void Enter(void)
    {
        int prev_s;
        do
        {
            prev_s = TestAndSet(&m_s, 0);
            if (m_s == 0 && prev_s == 1)
            {
                break;
            }
            // reluinquish current timeslice (can only
            // be used when OS available and
            // we do NOT want to 'spin')
            // ::Sleep(0);
        }
        while (true);
    }
        // Tries to enter the lock, returns 0
    // when mutex is already locked,
    // returns != 0 when success
    inline int TryEnter(void)
    {
        int prev_s = TestAndSet(&m_s, 0);
        if (m_s == 0 && prev_s == 1)
        {
            return 1;
        }
        return 0;
    }
    // Leaves or unlocks the mutex
    // (should only be called by lock owner)
    inline void Leave(void)
    {
        TestAndSet(&m_s, 1);
    }

    protected:
        // sets BIT value and returns previous
    // value in 1 atomic un-interruptable operation
    int TestAndSet(int* pTargetAddress, int nValue);

    private:
    int m_s;
};

    // This part is Platform dependent!
#ifdef WIN32
inline int CPP_SpinLock::TestAndSet(int* pTargetAddress,
                                                 int nValue)
{
    __asm
    {
        mov edx, dword ptr [pTargetAddress]
        mov eax, nValue
        lock xchg eax, dword ptr [edx]
    }
        // mov = 1 CPU cycle
      // lock = 1 CPU cycle
        // xchg = 3 CPU cycles
}
#endif // WIN32
```

3.2 Reentrant Spin Lock

Collapse

```
    // THIS LOCK IS REENTRANT !!
    // thanks to George V. Reilly for pointing out
    // the flaws in the first inefficient implementation
    // of mine (in the second one).
  // [illegible] is used as the Task, Thread or
    // Process Number, so windows you should
  // pass in GetCurrentThreadId() into it.
    // on embedded systems you [illegible] your owner
    // [illegible] (i.e. task or interrupt numbers).
class CPP_SpinLock_Reentrant_Hopefully_Less_Clumsy
{
public:
```

Attachment A

```
inline
  CPP_SpinLock_Reentrant_Hopefully_Less_Clumsy()
  m_nLockCount(0),
  m_nOwner(0xffffffff) {};

inline
  ~CPP_SpinLock_Reentrant_Hopefully_Less_Clumsy() {};

inline void Enter(unsigned int PNumber)
{
    if (PNumber == m_nOwner)
    {
        m_nLockCount++;
        return;
    }
    m_RealCS.Enter();
    m_nOwner = PNumber;
    m_nLockCount++;
    return;
}

inline int TryEnter(unsigned int PNumber)
{
    if (PNumber == m_nOwner)     [illegible]
    {
        m_nLockCount++;
        return 1;
    }
    if (m_RealCS.TryEnter())
    {
        m_nOwner = PNumber;
        m_nLockCount++;
        return 1;
    }
    return 0;
}

inline void Leave(unsigned int PNumber)
{
    assert(PNumber == m_nOwner);
    m_nLockCount--;
    if (m_nLockCount == 0)
    {
        m_nOwner = 0xffffffff;
        m_RealCS.Leave();
    }
}

protected:

private:
    CPP_SpinLock m_RealCS;
    unsigned int m_nLockCount;
    unsigned int m_nOwner;
};
```

4. Spin Lock Usage

4.1 When should you use a Spin Lock?

Only when it makes sense not to do a context switch in the event of a locked resource:

- The time the thread has to wait is smaller then the amount of time to do the context switch. (This is only true for SMP operating systems running on SMP hardware! If there is no real parallelism and you have a pre-emptive OS, you should always prefer a context switch. Which thread is going to free the resource anyway on a uni-processor system?)
- The chance of a resource conflict (= the chance of having to actually wait for the lock) in the critical section is either very small or the average waiting time is less then the time needed for the OS to do the context switching.
- The critical section itself is very small and the number of checks is very high.
- You don't have another synchronization mechanism because you do not have an OS.

4.2 Benefits of using a Spin Lock:

Depends on the situation, of course (as we will see in a couple of tests): **speed**. An NT Critical Section check (and the resource is 'free' - you have a "green light") takes about 6 CPU cycles. (I haven't checked this - it depends also on the type of CPU - but I remember reading it somewhere. N.B.: You can always do further testing yourself if you wanted...).

5. The tests

The test system is a dual PII 350 MHz (2 CPUs), 256 MB RAM and running Windows 2000 Advanced Server Edition.

Depending on the situation, the NOT-REENTRANT version is about 1 to 5 times faster (and remember it is sometimes also slower and wasting resources) than the `EnterCriticalSection/LeaveCriticalSection` stuff, which is the fastest Win32 synchronization mechanism. The REENTRANT version, however, is a lot (more than 2 times) slower and is only provided for no-OS Embedded systems programmers. I am sure the implementation can be a lot faster when using more 'assembler', but that would decrease the portability (A better re-entrant algorithm would also do the trick). So if you do not need re-entrance, do not use the re-entrant version!

In all the tests, the higher the number, the better the performance. There is no absolute proof to be found in these tests. It just helps you to reflect on things. Let common sense prevail.

5.1 Test 1:

This test code will generate a lot of collisions in the critical section, so other differences will emerge when collisions are sparse.

⊟ Collapse

```
 [illegible] value = 0;
 [illegible] prevvalue = 0;
 bool stop = false;

 CPP_SpinLock theTestMutex;
```

Attachment A

```
CRITICAL_SECTION theCS;

void IncreaseValue(unsigned int MNumber)
{
    theTestMutex.Enter();
    // EnterCriticalSection(&theCS);
    prevvalue = value;
    value++;
    if (prevvalue != value-1)
    {
        MessageBox(NULL, "Oh oh Oh!", 0, MB_OKCANCEL);
    }
    theTestMutex.Leave();
    // LeaveCriticalSection(&theCS);
}

DWORD WINAPI ThreadProc( LPVOID lpParameter )
{
    while (!stop)
    {
        IncreaseValue((unsigned int) lpParameter);
    }
    return 0;
}

int main(int argc, char* argv[])
{
    HANDLE hThread1 = NULL;
    HANDLE hThread2 = NULL;
    DWORD dwThreadID1 = 0;
    DWORD dwThreadID2 = 0;

    InitializeCriticalSection(&theCS);

    hThread1 = ::CreateThread(NULL, 0,
      ThreadProc, (void*) 0, 0, &dwThreadID1);
    hThread2 = ::CreateThread(NULL, 0, ThreadProc,
      (void*) 1, 0, &dwThreadID2);

    WaitForSingleObject(hThread1, 10000);

    stop = true;

    WaitForSingleObject(hThread1, INFINITE);
    WaitForSingleObject(hThread2, INFINITE);

    CloseHandle(hThread1);
    CloseHandle(hThread2);

    DeleteCriticalSection(&theCS);

    cerr << "\nNumber Increase:"
            << value << "\n";

    return 0;
}
```

```
Using the CRITICAL_SECTION
Run1 output: Number Increase:1.2227e+006
Run2 output: Number Increase:1.39654e+006
Run3 output: Number Increase: 1.33271e+006
Run4 output: Number Increase: 1.35419e+006

Using the Spin Lock
Run1 output: Number Increase: 8.43261e+006
Run2 output: Number Increase: 9.13326e+006
Run3 output: Number Increase: 8.1077e+006
Run4 output: Number Increase: 7.81813e+006
```

This certainly would prove my point, but don't cheer just yet. The performance differences are subtler than that.

5.2 Test 2:

Now, we change the test code, so we no longer have collisions on the lock:

Collapse

```
double value[2];
double prevvalue[2];
bool stop = false;

CPP_SpinLock theTestMutex[2];
CRITICAL_SECTION theCS[2];

void IncreaseValue(unsigned int MNumber)
{
    theTestMutex[MNumber].Enter();
    // EnterCriticalSection(&theCS[MNumber]);
    prevvalue[MNumber] = value[MNumber];
    (value[MNumber])++;
    if (prevvalue[MNumber] != value[MNumber]-1)
    {
        MessageBox(NULL, "Oh oh Oh!", 0, MB_OKCANCEL);
    }
    theTestMutex[MNumber].Leave();
    // LeaveCriticalSection(&theCS[MNumber]);
}

DWORD WINAPI ThreadProc( LPVOID lpParameter )
{
    while (!stop)
    {
        IncreaseValue((unsigned int) lpParameter);
    }
    return 0;
}
```

Attachment A

```
[illegible] main( [illegible] argc, [illegible] argv[])
{
    HANDLE hThread1 = NULL;
    HANDLE hThread2 = NULL;
    DWORD dwThreadID1 = 0;
    DWORD dwThreadID2 = 0;

    InitializeCriticalSection(&(theCS[0]));
    InitializeCriticalSection(&(theCS[1]));

    value[0] = 0;
    prevvalue[0] = 0;
    value[1] = 0;
    prevvalue[1] = 0;

    hThread1 = ::CreateThread(NULL, 0,
      ThreadProc, [illegible] 0, 0, &dwThreadID1);
    hThread2 = ::CreateThread(NULL, 0, ThreadProc,
      [illegible] 1, 0, &dwThreadID2);

    WaitForSingleObject(hThread1, 10000);

    stop = [illegible];

    WaitForSingleObject(hThread1, INFINITE);
    WaitForSingleObject(hThread2, INFINITE);

    CloseHandle(hThread1);
    CloseHandle(hThread2);

    DeleteCriticalSection(&(theCS[0]));
    DeleteCriticalSection(&(theCS[1]));

    cerr << "[illegible]"
      << value[0] + value[1] << "\n";

    [illegible] 0;
}
```

```
Using the CRITICAL_SECTION
Run1 output: Number increases: 1.01325e+007
Run2 output: Number increases: 1.06665e+007
Run3 output: Number increases: 1.13038e+007
Run4 output: Number increases: 1.04613e+007

Using the Spin Lock
Run1 output: Number increases: 1.04721e+007
Run2 output: Number increases: 1.13043e+007
Run3 output: Number increases: 1.17863e+007
Run4 output: Number increases: 1.09453e+007
```

In my opinion, there is hardly a significant difference, although the spin lock seems to do a slightly better job.

5.3 Test 3:

⊟ Collapse

```
[illegible] value;
[illegible] prevvalue;
[illegible] stop = [illegible];

CPP_SpinLock theTestMutex;
CRITICAL_SECTION theCS;

DWORD WINAPI ThreadProc2( LPVOID lpParameter )
{
    [illegible] (!stop)
    {
            [illegible]
        EnterCriticalSection(&(theCS));
        (value)++;
            [illegible]
        LeaveCriticalSection(&(theCS));
    }
    [illegible] 0;
}

[illegible] main([illegible] argc, [illegible] argv[])
{
    HANDLE hThread1 = NULL;
    DWORD dwThreadID1 = 0;

    InitializeCriticalSection(&(theCS));

    value = 0;
    prevvalue = 0;

    hThread1 = ::CreateThread(NULL, 0,
      ThreadProc2, [illegible] 0, 0, &dwThreadID1);

    WaitForSingleObject(hThread1, 10000);

    stop = [illegible];

    WaitForSingleObject(hThread1, INFINITE);

    CloseHandle(hThread1);

    DeleteCriticalSection(&(theCS));

    cerr << "[illegible]"
          << value << "[illegible]";

    [illegible] 0;
}
```

Attachment A

```
Using the CRITICAL_SECTION
Run1 output: Number Increases: 4.18735e+007
Run2 output: Number Increases: 4.18007e+007
Run3 output: Number Increases: 4.19747e+007
Run4 output: Number Increases: 4.2017e+007

Using the Spin Lock
Run1 output: Number Increases: 4.19765e+007
Run2 output: Number Increases: 4.1972e+007
Run3 output: Number Increases: 4.20036e+007
Run4 output: Number Increases: 4.19851e+007
```

This also does not show a 'big' difference (IMHO, it does not show a difference at all). I just think the Win32 Critical Sections have a pretty efficient implementation.

6. Conclusion

It's up to the potential user of these Spin Locks to decide which application is best suited for them.

Spin Locks can either be used on an embedded device without an operating system, or they can be used - with care - on a SMP machine with a SMP operating system. The mechanism is used to enforce mutual exclusion for shared resources. It is not recommended to use (blocking) Spin Locks on uni-processor systems. The key question is, whether the developer will let the scheduler intervene in case of a locked "shared resource" or "critical section". If it is decided to "spin" the lock, the developer must be certain that performance will be gained by not triggering the context switch.

History

- 24 Oct 2000 - Updated reentrant code in section 3.2

About Gert Boddaert

Studied applied economics but became seriously infected by the IT virus, so studied some more and eventually ended up as a software design engineer or system engineer, whatever sounds best.
Worked 'chronologically' at Agfa, KredietBank, Spemex, Xircom - an Intel Company. Now works for Sioux, technical software consulting. Wrote some stuff concerning software simulation of hardware, telecom management systems, modems, Bluetooth and drivers (WDM and others).
After work, works out (fitness), plays soccer, watches movies, drinks a 'trappist' beer and at the moment also tinkers with neural networks. Likes fast cars and is the proud owner of a 'moderately' powered Peugeot 306 GTI. Still tries to finish part III of Tolkien's famous book.
'nough said, back to work..

Click here to view Gert Boddaert's online profile.

Other popular Threads, Processes & IPC articles:

- Three Ways to Inject Your Code into Another Process
  How to inject code into another processes address space, and then execute it in the context of this process.
- Bridge Between Java And Windows
  Make your windows programs communicate with Java programs easily
- Using Worker Threads
  Learn how to create and use worker threads in your applications.
- RMI for C++
  User-friendly remote method invocation in C++.

[Top]

Sign in to vote for this article: Poor ○ ○ ○ ○ ○ Excellent

Note: You must Sign in to post to this message board.

FAQ

Message score threshold 3.0 Search comments

View Message View Per page 20

Msgs 1 to 21 of 21 (Total: 21) (Refresh)

First Prev Next

Subject	Author	Date
Problem with InitializeCriticalSectionAndSpinCount	eryreyeryeryeryeyryye	9:38 22 Mar '07
Atomic or not	Pierre Couderc	4:02 4 Jan '06
Implementing critical section with lists in multi threaded environment	vallabh inamdar	9:18 12 Apr '02
InitializeCriticalSectionAndSpinCount	Gerald Schwab	23:08 11 Oct '01
How about a context switch free reader writer lock?	Pieter Viljoen	23:53 7 Nov '00
Re: How about a context switch free reader writer lock?	Gert Boddaert	7:58 16 Nov '00
Re: How about a context switch free reader writer lock?	Pieter Viljoen	18:24 18 Nov '00
Re: How about a context switch free reader writer lock?	Gert Boddaert	9:13 24 Nov '00
Priority Inversion	Rob Krakora	8:00 26 Oct '00
Re: Priority Inversion	Gert Boddaert	[illegible]
What's about SetCriticalSectionSpinCount()	Daniel Lehmann	13:21 24 Oct '00
Re: What's about SetCriticalSectionSpinCount()	Gert Boddaert	[illegible]
Comments on the code	George V. Reilly, Microsoft	19 30 23 Oct '00